

Computing All Distinct Squares in Linear Time for Integer Alphabets

Hideo Bannai¹, Shunsuke Inenaga¹, and Dominik Köppl²

¹Department of Informatics, Kyushu University, Japan

²Department of Computer Science, TU Dortmund, Germany

October 12, 2016

Abstract

Given a string on an integer alphabet, we present an algorithm that computes the set of all distinct squares belonging to this string in time linear to the string length. As an application, we show how to compute the tree topology of the minimal augmented suffix tree in linear time. Besides from that, we elaborate an algorithm computing the longest previous table in a succinct representation using compressed working space.

1 Introduction

A square is a string of the form SS , where S is some non-empty string. It is well-known that a string of length n contains at most $n^2/4$ squares. This bound is the number of *all* squares, i.e., we count multiple occurrences of the same square, too. If we consider the number of all *distinct* squares, i.e., we count *exactly one* occurrence of each square, then it becomes linear in n : The first linear upper bound was given by Fraenkel and Simpson [11] who proved that the maximum number of the distinct squares a string of length n contains is upper bounded by $2n$. Later, Ilie [16] showed the slightly improved bound of $2n - \Theta(\lg n)$. Recently, Deza et al. [5] refined this bound to $\lfloor 11n/6 \rfloor$. In the light of these results one may wonder whether future results will “converge” to the upper bound of n : The *distinct square conjecture* is that the maximum number of distinct squares a string of length n contains is at most n . However, there still is a big gap between the best known bound and the conjecture. While studying a combinatorial problem like this, it is natural to think about ways to actually compute the exact number.

This article focuses on a computational problem on distinct squares, namely, we wish to compute (a compact representation of) the set of all distinct squares in a given string. Gusfield and Stoye [14] tackled this problem with an algorithm running in $\mathcal{O}(n\sigma_T)$ time, where σ_T denotes the number of different characters contained in the input text T of length n . Although its running time is optimal $\mathcal{O}(n)$ for a constant alphabet, it becomes $\mathcal{O}(n^2)$ for a large alphabet since σ_T can be as large as $\mathcal{O}(n)$.

We present an algorithm (Section 4) that computes this set in $\mathcal{O}(n)$ time for a given string of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$. Like Gusfield and Stoye, we can use the computed set to decorate the suffix tree with all squares (Section 5). As an application, we provide an algorithm that computes the tree topology of the minimal augmented suffix tree in linear time (Section 6). The best known algorithm to compute this tree topology takes $\mathcal{O}(n \lg n)$ time [2].

For our approach, we additionally need the longest previous factor table [3]. As a side result of independent interest, we show in Section 3 how to store this table in $2n + o(n)$ bits, and give an algorithm that computes it using compressed working space.

2 Definitions

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. An element w in Σ^* is called a **string**, and $|w|$ denotes its length. We denote the i -th character of w with $w[i]$, for $1 \leq i \leq |w|$. When w is represented by the concatenation of $x, y, z \in \Sigma^*$, i.e., $w = xyz$, then x , y and z are called a **prefix**, **substring** and **suffix** of w , respectively.

The **longest common prefix** of two strings is the number of characters of the longest prefix shared by both strings. The **longest common extension (LCE)** query asks for the longest common prefix of two suffixes of the (same) string.

A **factorization** of a string T is to partition T into non-empty substrings such that the concatenations of the substrings is T . Each substring is called a **factor**.

Our computational model is the word RAM model with word size $\Omega(\lg n)$. In the rest of this paper, we take a string T of length $n > 0$, and call it **the text**. We assume that $T[n] = \$$ is a special character that appears nowhere else in T , so that no suffix of T is a prefix of another suffix of T . We further assume that T is read-only; accessing a word costs $\mathcal{O}(1)$ time. We sometimes need the **reverse** of T , which is given by the concatenation $T[n-1] \cdots T[1] \cdot T[n]$.

We identify occurrences of substrings with their position and length in the text, i.e., if x is a substring of T , then there is a $1 \leq i \leq n$ and a $0 \leq \ell \leq n - i + 1$ such that $T[i..i + \ell - 1] = x$. In the following, we will represent the occurrences of substrings by tuples of position and length. When storing these tuples in a set, we call the set **distinct**, if there are no two tuples (i, ℓ) and (i', ℓ) such that $T[i..i + \ell - 1] = T[i'..i' + \ell - 1]$. A special kind of substring is a square: A **square** is a string of the form SS for $S \in \Sigma^+$; we call S and $|S|$ the **root** and the **period** of the square SS , respectively. Like with substrings, we can generate a set containing some occurrences of squares. A set of **all distinct squares** is a distinct set of occurrences of squares that is maximal under inclusion.

The **suffix tree** of T is the tree obtained by compacting the trie of all suffixes of T ; it has n leaves and at most n internal nodes. The leaf corresponding to the i -th suffix is labeled with i . Each edge e stores a string that is called the **label** of e . The **string label** of a node v is defined as the concatenation of all edge labels on the path from the root to v ; the **string depth** of a node is the length of its string label.

SA and ISA denote the suffix array and the inverse suffix array of T , respectively [19]. The access time to an element of SA is denoted by t_{SA} . LCP is an array such that $LCP[i]$ is the longest common prefix of $T[SA[i]..]$ and $T[SA[i-1]..]$ for $i = 2, \dots, n$. For our convenience, we define $LCP[1] := 0$.

A **range minimum query (RMQ)** asks for the smallest value in an integer array for a given range. There are data structures that can answer an RMQ on an integer array of length n in constant time while taking $2n + o(n)$ bits of space [9]. An LCE query for the suffixes $T[s..]$ and $T[t..]$ can be answered with an RMQ data structure on LCP with the range $[ISA[s] + 1..ISA[t]]$ in constant time.

A **bit vector** is a string on a binary alphabet. A **select query** on a bit vector asks the position of the i -th ‘0’ or ‘1’ in the bit vector. There are data structures that can be built in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ bits of working space such that they take $o(n)$ bits on top of the bit vector, and can answer the above query in constant time [21].

algorithm	time	working space	output space
Lemma 3.2,[4]	$\mathcal{O}(nt_{\text{SA}})$	$ \text{SA} + \text{LCP} + \mathcal{O}(\lg n)$	$n \lg n$
Corollary 3.3,[12, 15]	$\mathcal{O}(n \lg n)$	$n \lg n + 2n + o(n)$	$n \lg n$
Lemma 3.6,[18]	$\mathcal{O}(n/\epsilon^2)$	$(1 + \epsilon)n \lg n + \mathcal{O}(n)$	$2n + o(n)$
Lemma 3.6,[10]	$\mathcal{O}(nt_{\text{SA}})$	$\mathcal{O}(n \lg \sigma)$	$2n + o(n)$

Table 1: Algorithms computing LPF; space is counted in bits. The output space is not considered as working space. $0 < \epsilon \leq 1$ is a constant.

3 A Compact Representation of the LPF Array

The array LPF, called *longest previous factor table* of T in the literature, is formally defined as

$$\text{LPF}[j] := \max \{ \ell \mid \text{there exists an } i \in [1..j-1] \text{ such that } T[i, i+\ell-1] = T[j, j+\ell-1] \}.$$

It is useful for computing the **Lempel-Ziv factorization** of $T = f_1 \cdots f_z$, which is defined as $f_i = T[k..k + \text{LPF}[k]]$ with $k := \sum_{j=1}^{i-1} |f_j| + 1$ for $1 \leq i \leq z$.

Corollary 3.1. *Given LPF, we can compute the Lempel-Ziv factorization in $\mathcal{O}(n)$ time. We can represent it by a bit vector of length n , in which we mark the factor beginnings. A select data structure on top of the bit vector allows access to the i -th factor in constant time.*

Since the algorithm in Section 4 needs LPF, we are interested in the time and space bounds for computing LPF. We start with the (to the best of our knowledge) state of the art algorithm with respect to time and space requirements.

Lemma 3.2 ([4, Theorem 1]). *Given SA and LCP, we can compute LPF in $\mathcal{O}(nt_{\text{SA}})$ time. Besides the output space of $n \lg n$ bits, we only need constant working space.*

Besides this algorithm, we are only aware of some practical improvements [20, 17].

Let us consider the size of LCP needed in Lemma 3.2. Sadakane [22] showed a $2n + o(n)$ -bits representation of LCP. Thereto he stores the array PLCP defined as $\text{PLCP}[\text{SA}[i]] = \text{LCP}[i]$ in a bit vector (assume $\text{LCP}[1] = 0$) in the following way (also described in [8]): Since $\text{PLCP}[1] + 1, \text{PLCP}[2] + 2, \dots, \text{PLCP}[n] + n$ is a non-decreasing sequence with $1 \leq \text{PLCP}[1] + 1 \leq \text{PLCP}[n] + n = n$ ($\text{PLCP}[i] \leq n - i$ since the terminal \$ is a unique character in T) the values $I[1] := \text{PLCP}[1]$ and $I[i] := \text{PLCP}[i] - \text{PLCP}[i-1] + 1$ ($2 \leq i \leq n$) are non-negative. By writing $I[i]$ subsequently for each $2 \leq i \leq n$ in unary code $0^{I[i]}1$ to a bit vector S , we can compute $\text{PLCP}[i] = \text{select}_1(S, i) - 2i$ and $\text{LCP}[i] = \text{select}_1(S, \text{SA}[i]) - 2\text{SA}[i]$. Moreover, $\sum_{i=1}^n I[i] \leq n$ and therefore S is of length at most $2n$.

By using Sadakane's LCP-representation, we get LPF with the algorithm of Crochemore et al. [4] in the following time and space bounds:

Corollary 3.3. *Having SA and LCP stored in $n \lg n$ bits (this allows $t_{\text{SA}} = \mathcal{O}(1)$) and $2n + o(n)$ bits, respectively, we can compute LPF with $\mathcal{O}(\lg n)$ additional bits of working space (not counting the space for LPF) in $\mathcal{O}(n)$ time.*

By plugging in a suffix array construction algorithm like the in-place construction algorithm by Franceschini and Muthukrishnan [12], we get the bounds shown in Table 1 (since we can build LCP in-place after having SA [15]).

Although this result seems compelling, this approach stores the suffix array and LPF in plain arrays (the former for getting constant time access). In the following, we will show that the LPF array can be stored more compactly. To this end, we elaborate two algorithms, one that runs in nearly linear time using $\mathcal{O}(n \lg \sigma)$ bits of space. Unfortunately, we have to come up with a new approach since this algorithm creates a plain array to get constant time random write-access for computing the entries of LPF.

We start with the new representation of LPF, for which we use the same trick as for PLCP due to the following property (which is crucial for squeezing PLCP into $2n + o(n)$ bits).

Lemma 3.4. $n - j \geq \text{LPF}[j] \geq \text{LPF}[j - 1] - 1$ for $2 \leq j \leq n$.

Proof. By definition there is a $1 \leq i < j - 1$ such that $T[i, i + \text{LPF}[j] - 1] = T[j - 1, j - 1 + \text{LPF}[j] - 1]$. Hence $T[i + 1, i + \text{LPF}[j] - 1] = T[j, j - 1 + \text{LPF}[j] - 1]$. \square

We conclude that the sequence $\text{LPF}[1] + 1, \text{LPF}[2] + 2, \dots, \text{LPF}[n] + n$ is non-decreasing with $1 \leq \text{LPF}[1] + 1 \leq \text{LPF}[n] + n \leq n$. We immediately get:

Corollary 3.5. LPF can be represented by a bit vector with select-support such that access is in constant time. The data structures use $2n + o(n)$ bits.

Now we present the two algorithms that compute LPF in this representation with the aid of the suffix tree. Both algorithms are derivatives of the algorithms [18, 10] that compute the Lempel-Ziv factorization. We aim at building the LPF-representation of Corollary 3.5 directly such that we do not need to allocate the plain LPF array using $n \lg n$ bits in the first place. To this end we create a bit vector of length $2n$ and store in it the LPF values successively. In more detail, we follow the description of the Lempel-Ziv factorization algorithms presented in [18, 10]. There, the algorithms are divided into several passes. In each pass we successively visit leaves in text order (determined by the labels of the leaves). We only have to do a single pass and stop after it. Similarly to the first passes of the two Lempel-Ziv algorithms, we use a bit vector B_V to mark already visited internal nodes. When visiting a leaf we climb up the tree until reaching the root or an already marked node. In the former case (we climbed up to the root) we output zero. In the latter case, we output the string depth of the marked node. By doing so, we have computed $\text{LPF}[1..j]$ after the j -th leaf-to-root traversal.

Lemma 3.6. We can compute LPF in $\mathcal{O}(nt_{\text{SA}})$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space, or in $\mathcal{O}(n/\epsilon^2)$ time using $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, for a constant $0 < \epsilon \leq 1$. Both variants include the space of the output in their working spaces.

Proof. Computing the string depth of a node involves access to an RMQ data structure of LCP, and an access to SA. Both accesses can be emulated by the compressed suffix array in t_{SA} time, given that we have computed LCP in the above representation. \square

Unless there is a linear time algorithm that can construct and store SA and LCP in less than $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ space, the linear time result of this lemma is more efficient than the result of Lemma 3.2.

4 Computing the Set of All Distinct Squares

Given a string T , our goal is to compute all distinct squares of T . Thereto we return a set of pairs consisting of a starting position s and a length ℓ such that $(s, \ell) \mapsto T[s..s + \ell - 1]$ is the leftmost occurrence of a square. The size of this set is linear due to

Lemma 4.1 (Fraenkel and Simpson [11]). *A string of length n can contain at most $2n$ distinct squares.*

We follow the approach of Gusfield and Stoye [14]. Their idea is to compute a set of squares (the set stores pairs of position and length like described in Section 2)¹ with which they can generate all distinct squares. They call this set of squares a **leftmost covering set**. A leftmost covering set obeys the property that the leftmost occurrence of every square (k, ℓ) can be constructed by **right-rotating** a square of this set with the operation $(i, \ell) \mapsto T[i + j..i + \ell + j - 1] = T[k..k + \ell - 1] = T[i + j..i + \ell - 1]T[i..i + j - 1]$ (for some j such that $i + j = k$ and the right hand equation holds). We call a leftmost covering set S **minimal** if there is no strict subset of S such that the above property holds. Unfortunately, the leftmost covering set computed in [14] is not necessarily minimal since it does not have to be distinct.

We are now ready to present our modification of the algorithm of [14]. To this end, we briefly review how their approach works: They compute their leftmost covering set by examining the borders between all Lempel-Ziv factors $f_1 \cdots f_z = T$. That is because

Lemma 4.2 ([14, Theorem 5]). *The leftmost occurrence of a square $(i, 2p) \mapsto T[i..i + 2p - 1]$ touches at least two Lempel-Ziv factors. Let f_x be the factor that contains the center of the square $i + p - 1$. Then either*

- (a) *the square has its left end (position i) inside f_x and its right end (position $i + 2p - 1$) inside f_{x+1} ,
or*
- (b) *the left end of the square extends into f_{x-1} (or even further left). The right end can be contained inside f_x or f_{x+1} .*

Having a data structure for computing LCE queries on the text and on its inverse, they can probe at the borders of two consecutive factors whether there is a square. Roughly speaking, they have to check at most $|f_x| + |f_{x+1}|$ many periods at the borders of every two consecutive factors f_x and f_{x+1} due to the above lemma. This gives $\sum_{x=1}^z t_{\text{LCE}}(|f_x| + |f_{x+1}|) = \mathcal{O}(nt_{\text{LCE}})$ time, during which they can compute a leftmost covering set L . Subsequently, their algorithm runs the so-called Phase II. The task of this phase is to compute all distinct squares with the aid of the suffix tree. It begins with computing the locations of the squares in a subset $L' \subseteq L$ in the suffix tree in $\mathcal{O}(n)$ time. This subset L' is still guaranteed to be a leftmost covering set. Finally, their algorithm computes all distinct squares of the text by right-rotating the squares in L' . In their algorithm, the right-rotations are done by *suffix link walks* over the suffix tree. Their running time analysis is based on the fact that each node has at most σ_T incoming suffix links, where σ_T denotes the number of different characters occurring in the text T . Given that the number of distinct squares is linear, Phase II runs in $\mathcal{O}(n\sigma_T)$ time.

In the following, we will present our modification of the above sketched algorithm. To speed up the computation, we discard the idea of using the suffix links for right-rotating squares (i.e., we skip Phase II completely). Instead, we use LPF to filter already found squares. By doing so, there are only distinct squares in the filtered set. Additionally, we compute all distinct squares by right-rotating the squares belonging to this filtered (and hence minimal) leftmost covering set. In more detail, we modify Algorithm 1 of [14] by filtering the reported squares in the following way (see Algorithm 1): Given the starting position s and the period p of a square returned by Algorithm 1 in [14], we consider the square $T[s..s + 2p - 1]$ and its right rotations as candidates of our list. Thereto we check whether $\text{LPF}[s + j]$ is less than $2p$, successively for each $0 \leq j \leq p$ until $T[s + j..s + j + 2p - 1]$ is not a square (end of a run)

¹It differs to the set we want to compute by the fact that they allow, among others, non-distinct in their set.

or $\text{LPF}[s+j] \geq 2p$ holds. Assume that $T[s+j..s+j+2p-1]$ is a square. If $\text{LPF}[s+j] < 2p$, we report this square. Otherwise ($\text{LPF}[s+j] \geq 2p$), this square is not the leftmost occurrence. Hence, we can omit $T[s+j..s+j+2p-1]$. We can further skip checking $j+1, \dots, p$, since a square $T[s+k..s+k+2p-1]$ (for some $k \in [j+1..p]$) is already reported ($\text{LPF}[s+k] \geq 2p$) or its leftmost occurrence starts after s and thus will be found later.

Theorem 4.3. *Given an LCE data structure with $\mathcal{O}(t_{\text{LCE}})$ access time and LPF, we can compute all distinct squares in $\mathcal{O}(nt_{\text{LCE}} + \text{occ}) = \mathcal{O}(nt_{\text{LCE}})$ time, where occ is the number of distinct squares.*

Proof. We show that the returned list is the list of all distinct squares. No square occurs in the list twice since we only report the occurrence of a square (i, ℓ) if $\text{LPF}[i] < \ell$. Assume that there is a square S missing in the list. Let $T[i..i+\ell-1] = S$ be its leftmost occurrence. There is a leftmost square $R = T[j..j+\ell-1]$ reported by the (original) algorithm of Gusfield and Stoye [14] such that $i - \ell/2 < j \leq i$ and right-rotating R yields S ; let j be the largest such value. Then R is the leftmost occurrence; hence R is contained in our returned list (since we capture all leftmost occurrences of the squares returned by the original algorithm). For all $j \leq k \leq i$ it holds that $\text{LPF}[k] < \ell/2$ (otherwise j is not the largest possible value). Hence, we have reported S .

The occ term in the running time is dominated by the nt_{LCE} term due to Lemma 4.1. \square

5 Decorating the Suffix Tree with All Squares

We adopt the idea of Gusfield and Stoye for representing the set of all distinct squares by a decoration of the suffix tree. The idea is to mark the squares in the suffix tree. More precisely, our task is to compute a set of tuples of the form $(\text{node}, \text{length})$ such that a tuple storing a node v and a length ℓ represents the square that is the (not necessarily proper) prefix of the string label of v of length ℓ . Fortunately, we can follow some parts of the above described Phase II to compute this list. Thereto we need the reported squares to be grouped by their starting positions; this is done by modifying our algorithm computing all distinct squares in the following way: We create a queue storing the lengths of the squares starting at the position i , for each $1 \leq i \leq n$; we further want the contents of the queues to be sorted in descending order. We can fill the queues *without* sorting by iterating over the period length in the outer loop, and iterating over all Lempel-Ziv factors in the nested loop. Since we iterate over all factors for all possible periods, we want to skip small factors when they become shorter than the period (that is incremented in the outer loop). We do this by adding an additional array Z of $z \lg z$ bits that is zero initialized. The array Z stores for each insufficiently large factor f_x in $Z[x]$ the index of the next factor whose length is sufficiently large. If $Z[x] \neq 0$, we skip all factors f_y with $y \in [x..Z[x]-1]$ whose length is smaller than the current period. This allows us running the computation in linear time.

We have to show that the computation still computes the minimal leftmost covering set. To this end, let us fix the period p (over which we iterate in the outer loop). By [14, Lemma 7], processing squares satisfying Lemma 4.2(a) before processing squares satisfying Lemma 4.2(b) (all squares have the period p) produces the desired output for period p . By iterating over all periods in ascending order, we can fill the queues as intended.

The values in the queues (i.e., the lengths of the squares starting at a specific position) can be stored in Elias-Fano coding [7, 6]. A little modification of this coding is necessary since it requires a list of ascendingly sorted integers (our queues start with the largest lengths); this can be done by mapping every length ℓ to $n - \ell$ (and hence inverting the order) internally. If the queue corresponding to the i -th leaf

($1 \leq i \leq n$) stores m_i elements, then $2occ + \sum_{k=1}^n (m_i \lceil \lg(n/m_i) \rceil) + o(occ)$ bits are needed to represent all queues. It is easy to implement the popping of the first value from a queue with this representation.

Finally, we can conduct Phase II described in [14]. In the original version, the goal of Phase II was to decorate the suffix tree with the endpoints of a subset of the previously computed leftmost covering set. We will show that doing exactly the same operations with the minimal leftmost covering set leads to the decoration of all squares in the suffix tree directly. In more detail, we associate the queue of those squares starting at the i -th position with the suffix tree leaf having label i , for each $1 \leq i \leq n$. We follow [14] by processing every node of the suffix tree with a bottom-up traversal. An internal node inherits the queue of the child whose subtree contains the leaf with the smallest label. If the edge to the parent node contains the ending position of one or more squares in the queue (they are all stored at the front of the queue), we decorate the edge with these squares, and pop them off from the queue. By [14, Theorem 8], there is no square of the set L' (defined in Section 4) neglected during the bottom-top traversal. The same holds if we exchange L' with our computed set of all distinct squares:

Lemma 5.1. *Feeding the algorithm of Phase II with the above constructed queues containing occurrences of squares belonging to the minimal leftmost covering set, it will decorate the suffix tree with all distinct squares.*

Proof. We show that no square of the input set is left out during the bottom-up traversal. Let us take a suffix tree node u with its children v and w . Without loss of generality, assume that the smallest label among all leaves contained in the subtree of v is smaller than the label of every leaf contained in w 's subtree. For the sake of contradiction, assume that the queue of w contains the occurrence of a square (i, ℓ) at the time when we pass the queue of v to its parent u . The length ℓ is smaller than v 's string depth, otherwise it would already have been popped off from the queue. But since v 's subtree contains a leaf whose label j is the smallest among all labels contained in the subtree of w , the square occurs before at $T[j..j + \ell - 1] = T[i..i + \ell - 1]$, a contradiction to the minimality of the minimal leftmost covering set. \square

This concludes the correctness of the modified algorithm. We immediately get:

Theorem 5.2. *Given LPF, an LCE data structure on the reversed text, and the suffix tree of T , we can decorate the suffix tree with all squares of the text in $\mathcal{O}(nt_{\text{LCE}})$ time. Besides from these data structures, we use $n \lg n + n + o(n)$ bits of working space.*

Proof. We need $n \lg n$ bits for storing the queues. We further store in a bit vector of length n with select-support the starting positions of the Lempel-Ziv factors such that accessing the i -th factor is done in constant time. LCE queries on the text can be answered by lowest common ancestor queries on the suffix tree; most representations allow this in constant time. \square

Corollary 5.3. *We can compute the suffix tree and decorate it with all squares of the text in $\mathcal{O}(n/\epsilon^2)$ time using $(1 + \epsilon)2n \lg n + \mathcal{O}(n)$ bits.*

Proof. We use $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits to store SA, ISA, LCP, LPF, and an RMQ data structure on LCP such that LCE queries can be answered in constant time. We additionally need the suffix array, its inverse, and the LCP array (with an RMQ data structure) of the reversed text to answer LCE queries on the reversed text. \square

Algorithm 1: Modified Algorithm 1 of [14]

```
1  $\mathbf{b}(f)$  denotes the left end of a factor  $f = T[\mathbf{b}(f)..\mathbf{b}(f) + |f| - 1]$ ,  $lcp$  and  $lcs$  compute the LCE in  $T$ 
   and the LCE in the reverse of  $T$  (mirroring the input indices by  $i \mapsto n - i$  for  $1 \leq i \leq n - 1$ ),
   respectively.
2 Assume  $\mathbf{b}(f_{z+1}) = n$ 
3 Let  $f_1, \dots, f_z$  be the factors of the Lempel-Ziv factorization
4  $Z \leftarrow$  array of size  $z \lg z$  bits, initialized with 0
5  $m \leftarrow \max(|f_1| + |f_2|, \dots, |f_{z-1}| + |f_z|)$ 
6 for  $p = 1, \dots, m$  do
7   for  $x = 1, \dots, z$  do
8     if  $|f_x| + |f_{x+1}| < p$  then
9        $y \leftarrow x$ 
10      while  $|f_y| + |f_{y+1}| < p$  do
11        if  $Z[y] \neq 0$  then  $y \leftarrow Z[y]$  else incr  $y$ 
12       $Z[x] \leftarrow y$ 
13     $x \leftarrow y$ 
14  if  $|f_x| \geq p$  then // probe for squares satisfying Lemma 4.2(a)
15     $q \leftarrow \mathbf{b}(f_{x+1}) - p$ 
16     $\ell_1 \leftarrow lcp(\mathbf{b}(f_{x+1}), q)$ 
17     $\ell_2 \leftarrow lcs(\mathbf{b}(f_{x+1}) - 1, q - 1)$ 
18    if  $\ell_1 + \ell_2 \geq p$  and  $\ell_1 > 0$  then // found a square of length  $2p$  with its right
      end in  $f_{x+1}$ 
19       $s \leftarrow \max(q - \ell_2, q - p + 1)$  // square starts at  $s$ 
20      if  $\text{LPF}[s] < 2p$  then report( $s, 2p$ ) for  $j = 1, \dots, p - 1$  do
21        if  $T[s + j - 1] \neq T[s + j + 2p - 1]$  or  $\text{LPF}[s + j] \geq 2p$  then break
        report( $s + j, 2p$ )
22     $q \leftarrow \mathbf{b}(f_x) + p$  // probe for squares satisfying Lemma 4.2(b)
23     $\ell_1 \leftarrow lcp(\mathbf{b}(f_x), q)$ 
24     $\ell_2 \leftarrow lcs(\mathbf{b}(f_x) - 1, q - 1)$ 
25     $s \leftarrow \max(\mathbf{b}(f_x) - \ell_2, \mathbf{b}(f_x) - p + 1)$  // square starts in a factor preceding  $f_x$ 
26    if  $\ell_1 + \ell_2 \geq p$  and  $\ell_1 > 0$  and  $s + p < \mathbf{b}(f_{x+1})$  and  $\ell_2 > 0$  then // found a square of
      length  $2p$  whose center is in  $f_x$ 
27      if  $\text{LPF}[s] < 2p$  then report( $s, 2p$ ) for  $j = 1, \dots, p - 1$  do
28        if  $T[s + j - 1] \neq T[s + j + 2p - 1]$  or  $\text{LPF}[s + j] \geq 2p$  then break report( $s + j, 2p$ )
```

6 Computing the Tree Topology of the MAST in Linear Time

A modification of the suffix tree is the *minimal augmented suffix tree (MAST)* [1]. This tree can swiftly answer the number of non-overlapping occurrences of a substring in T . To this end, it adds some nodes on the unary paths of the suffix tree, and stores the number of non-overlapping occurrences of the string label of each node. The newly created nodes obey the property that the stored numbers of the MAST nodes on the path from a leaf to the root are *strictly* increasing. Given a query string, we traverse the MAST from the root downwards while reading the query string from the edge labels. We stop at an edge whose in-going node v has the property that the query string is a (not necessarily strict) prefix of v 's string label. Then v 's stored number is the right answer. The MAST can be built in $\mathcal{O}(n \lg n)$ time [2].

In this section, we show how to compute the tree topology of the MAST in linear time. The topology of the MAST differs to the suffix tree by the fact that the root of each square is the string label of a

node. We show that we can compute a list storing the information about where to insert the new nodes. The list stores tuples consisting of a node v and a length ℓ ; we use this information later to create a new node w splitting the edge (u, v) into (u, w) and (w, v) , where u is the (former) parent of v . Further, we will label (w, v) with the last ℓ characters and (u, w) with the rest of the characters of (u, v) .

Our idea is to explore the suffix tree with a top-down traversal while locating the roots of the squares in the order of their lengths. In order to locate the roots of the squares in linear time we use two data structures. The first one is a semi-dynamic lowest marked ancestor data structure [13]. It allows marking a node and querying for the lowest marked ancestor of a node in constant amortized time. We will use it to mark the area in the suffix tree that is already processed for finding the roots of the squares. By doing so, we will visit a node as many times as we have to insert nodes on its in-going edge, plus one (this gives $\mathcal{O}(n + \text{occ}) = \mathcal{O}(n)$ time).

The second data structure is a sorted list consisting of the length of a square and its corresponding node. A node corresponds to a square S if it is the highest node whose string label has S as a (not necessarily proper) prefix. This list can be obtained by a simple DFS after decorating the suffix tree (Section 5). We further sort the list with respect to the square lengths with a linear time integer sorting algorithm.

Finally, we explain the algorithm locating the roots of all squares. We successively process all tuples of the list, starting with the shortest square length. Given a tuple of the list containing the node v and the length ℓ , we want to split an edge on the path from the root to v and insert a new node whose string depth is $\ell/2$. To this end, we compute the lowest marked ancestor u of v . If u 's string depth is smaller than $\ell/2$, we mark all descendants of u whose string depth is smaller $\ell/2$, and additionally the children of those nodes (this can be done by a DFS or a BFS). If we query for the lowest marked ancestor of u again, we get an ancestor w whose string depth is at least $\ell/2$, and whose parent has a string depth less than $\ell/2$. We report w and the subtraction of $\ell/2$ from w 's string depth. Since we never visit marked nodes twice we get linear time overall.

If the suffix tree is in a pointer-based representation, it is easy to add the new nodes by splitting the edges whose in-going nodes are contained in the output of our algorithm.

Lemma 6.1. *We can compute the tree topology of the MAST in linear time using linear number of words.*

7 Open Problems

It is left open to compute the number of non-overlapping occurrences of the string labels of the MAST nodes in linear time.

Acknowledgements

This work was mainly done during a visit at the Kyushu University in Japan, supported by the *Japan Society for the Promotion of Science (JSPS)*.

References

- [1] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.

- [2] G. Brodal, R. Lyngsø, A. Östlin, and C. Pedersen. Solving the string statistics problem in time $\mathcal{O}(n \log n)$. In *Automata, Languages and Programming*, volume 2380 of *LNCS*, pages 728–739. Springer, 2002.
- [3] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.
- [4] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. LPF computation revisited. In *Proc. IWOCA*, pages 158–169, 2009.
- [5] A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015.
- [6] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974. ISSN 0004-5411.
- [7] R. M. Fano. On the number of bits required to implement anassociative memory. *Memorandum 61, Computer Structures Group, Project MAC*, 1971.
- [8] J. Fischer. Wee LCP. *Inform. Process. Lett.*, 110(8–9):317–320, 2010.
- [9] J. Fischer and V. Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
- [10] J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.
- [11] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998.
- [12] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. ICALP*, volume 4596 of *LNCS*, pages 533–545. Springer, 2007.
- [13] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.
- [14] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.
- [15] W.-K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *Proc. CPM*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002.
- [16] L. Ilie. A note on the number of squares in a word. *Theor. Comput. Sci.*, 380(3):373–376, 2007.
- [17] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.
- [18] D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*. IEEE Computer Society, 2016, to appear.
- [19] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

- [20] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proc. CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2011.
- [21] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. *ACM Trans. Algorithms*, 3(4):Article No. 43, 2007.
- [22] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007.